Computer Science • Vol. 8 Special Edition • 2007

Przemysław Maciołek*, Paweł Król**, Jarosław Koźlak***

PROBABILISTIC ANOMALY DETECTION BASED ON SYSTEM CALLS ANALYSIS

We present an application of probabilistic approach to the anomaly detection (PAD). By analyzing selected system calls (and their arguments), the chosen applications are monitored in the Linux environment. This allows us to estimate "(ab)normality" of their behavior (by comparison to previously collected profiles). We've attached results of threat detection in a typical computer environment.

Keywords: *anomaly detection, IDS, system calls, Linux*

PROBABILISTYCZNE ROZPOZNAWANIE ANOMALII BAZUJĄCE NA ANALIZIE WYWOŁAŃ SYSTEMOWYCH

W artykule przedstawiono zastosowanie propabilistycznego podejścia do rozpoznawania anomalii (PAD). Poprzez analizę wybranych wywołań systemowych (oraz ich argumentów), monitorowane są aplikacje działające pod kontrolą Linux. Pozwala to na oszacowanie (a)normalności ich zachowania (poprzez porównanie z poprzednio zebranymi profilami). Załączone są rezultaty rozpoznawania zagrożeń w typowym środowisku komputerowym.

Słowa kluczowe: *rozpoznawanie anomalii, IDS, wywołania systemowe, Linux*

1. Introduction

Today, nobody doubts that the ubiquitous Internet has its "dark" side. Each connected computer can be a target of potential attack. It is impossible to make the software perfect and immediately fix all the existing "holes". Even if that would have been possible, many attacks actually succeed because of inappropriate configuration of server services (with use of weak passwords for instance).

But when all protections fail, there is still hope left – the Intrusion Detection Systems (IDS). The IDS's purpose is to track various aspects of the computer system or infrastructure and find any signs of security breaches. Currently they are an indispensable element of a good security policy.

* Ph.D. Student EAIiE, AGH-UST, Kraków, Poland, pmm@agh.edu.pl
** EAIiE, AGH-UST, Kraków, Poland, paulkrol@gmail.com
*** Institute of Computer Science, AGH-UST, Kraków, Poland, kozlak@agh.edu.pl

In this paper, we present an implementation of Host-Based, Anomaly type IDS. It is based on a fact that when given application (e.g. a server service) is being seized, it begins to act differently and its behaviour deviates from a specific norm. The Host-Based IDS (that is operating on a given machine) is tracing the applications, usually by tracking its system calls (they provide communication between kernel and processes running in the user space – e.g. if a program wants to read an open file, it executes a `read()` system call). Using special algorithms, this information is used to create an appropriate model, so the current behaviour could be compared to the well-known (previously collected) "normal" activity. A deviation between them is a sign of a possible policy violation.

Our goal was to create an IDS system that would achieve high intrusion detection rate, while maintaining low number of false alerts. We have extended a Probabilistic Anomaly Detection algorithm [15] with novel use of information carried by system calls arguments, thus the SysPAD was born. Most of host based anomaly IDS's use information related to system calls and their sequence. We have decided to focus on system calls arguments rather than the sequences, so we could understand not only what kind of system call has been requested by an application but also where and how it is going to operate.

Our implementation consists of two main parts – a Linux kernel patch that provides system calls tracing (we have used custom solution rather than `ptrace`) and a program that communicates with kernel and analyzes the data.

The system performance will be evaluated against real-life attacks. We will choose few commonly used applications with known flaws, observe their behavior during normal operation and then try to breach the security, using appropriate exploits. System rating will be estimated on the basis of number of actually detected attack attempts and number of correct behaviors classified as anomalies. In an ideal situation, we hope to see all attack attempts detected and no false alarms.

2. State of the arts in Host Based Anomaly IDS systems

A typical Host-Based Anomaly IDS program flow consists of two main phases:

1. Learning of the "normal" program behaviour. This is usually done once for a given software version.
2. Comparing current system state to previously gathered model of "normal" program behaviour.

Creating the "right" model and teaching an appropriate behavior is not an easy task. On the one hand – all probable behaviors of monitored application must occur (so we will be able to distinguish the correct events from the bad ones). On the other hand – improper events must not arise (otherwise an inappropriate behavior will be considered as a correct one).

2.1. Short sequences method – SEQ

Short sequences method [10], created at the New Mexico University in 1996, was one of the first attempts to create an IDS system leveraging the system call information. Nowadays, the method is mainly used as a point of reference during examination of various new anomaly detection projects.

During the learning phase the IDS monitors system calls invoked by a traced program. Based on them, a database containing all sequences of given length (typically 3 consecutive calls) is built.

During the detection phase, the IDS system that is monitoring the application traces its system calls and creates sequences of a given length. If a sequence is not present in a previously created database then it is considered as an anomaly.

2.2. Finite state automata – FSA

The Finite State Automata method – FSA was presented in 2001, during an IEEE conference [1]. Similarly to the SEQ, application system calls are monitored. Basing on retrieved information, a finite state automata (that is supposed to reflect the process behavior) is constructed.

System calls sequences itself does not contain direct information about internal automata states. Because of that, the PC counter *(Program Counter)* is additionally used. Data are stored in a graph, in which vertexes are defined by system call occurrence addresses and curves describing number of a specific system call.

Anomaly detection with this method is similar to the algorithm presented before. First – the system learns "standard" program behavior. Secondly, the detection phase starts. It is similar to the learning phase, but this time instead of adding new graph states, the algorithm verifies if a transition between two given states (going through specified system call) is allowed.

2.3. Virtual Paths – VtPath

The VtPath [2] method was created at the Universities of Massachusetts and Georgia, USA. It utilizes not only system call information, but also related function return addresses. VtPath (through the stack analysis) uses all consecutive function return addresses, starting from the system call itself.

During the learning phase, a list of all successive function return addresses is created for observed system calls. The list is in following form: $A_n = (a_0, a_1, \ldots, a_{n-1}, a_n)$, where n is a total number of stack frames and a_n is a return address from the function that actually made the system call.

There is a number of situations that inform about possible anomalies. For example, when it is impossible to retrieve the function return addresses from the stack trace (what happens during the *buffer overflow* attacks) or if element a_n in the addresses list is pointing to other system call.

2.4. Critical system elements access analysis, PAD

A slightly different approach to break-ins detection is based on monitoring especially important (from the security point of view) system elements. In case of MS Windows, the Registry (where information required by the system and its components is stored) might be considered as such an element. In case of Unix-like systems, the whole file system area might be treated as especially important, but practically most important information is usually stored in the `/etc` and `/var` directories.

Papers [7] and [14] present anomaly detection results, where Probabilistic Anomaly Detection (PAD) [15] algorithm was used. It uses a feature vector as a unit of information, representing single observed event.

3. Theoretic model of the SysPAD system

There are two major groups of IDS systems. The first one tries to analyze the program flow (i.e. using system calls information) and the latter is looking in other places for estimating the state of the system.

The first group of systems is supposed to show what program is currently doing. E.g. `open`, `write`, `close` – for opening a file, writing some data and closing it. However, we do not actually know what particular file was opened, as such systems do not collect this information.

On the other hand, the systems such as PAD (see s. 2.4) are not event based, but rather periodically scan various elements of computer system. Even if they find that the file was modified, they are not able to tell which application did it and in what sequence of operations.

We see a solution to these problems – an extensive use of information about system calls – not only what particular system call was observed and in what sequence, but rather what were its arguments.

In comparison to previously presented IDS systems, the SysPAD algorithm is extending the scope of retrieved data. It collects (and to some degree – transforms) information about the system call arguments. This is in oppose to methods such as SEQ or VtPath, which use only system call number (and sometimes some other assistant data – like PC counter or function return address). While those data allowed to model the traced application program flow, our approach makes it possible to finely analyze actual program operations.

3.1. Feature Vectors

The IDS, tracing program execution, monitors all induced system calls and represents them as a list of vectors. Each vector contains a number of selected features and from the algorithm point of view is considered as a single event – a system call.

Chosen features are listed in Table 1.

Table 1
Data model used in the system

No.	Feature
1	process identifier
2	system call id
3	first system call argument
4	second system call argument
5	third system call argument

While some system calls take many arguments, it was decided to use at most only three of them, as the rest was usually too complicated to analyze or wasn't contributing useful data from our point of view.

3.2. Consistency Checks

The probabilistic approach to anomaly detection might be reduced to problem of the density function $P(x)$ calculation (which describes probability of vector features occurrence) on set of data we are interested in. If we are able to estimate such function on data representing the process behavior, then we will be able to define untypical behavior (an anomaly) as an event that occurs with a very small probability.

Density function estimation on such set of data is not an easy task. For practical reasons, the approach was slightly modified. The gathered data undergo a series of consistency tests. If at least one of the tests is unsuccessful, the tested information is considered as an anomaly.

First order consistency checks – the features of observed vectors are checked for compliance with features in previously collected profiles. In other words – we check if any single feature from a vector is already known. Probability of observed element occurrence will be noted as $P(X_i)$.

First order consistency checks are able to identify previously unknown calls, processes or system call arguments.

Second order consistency checks – the feature vector elements are permuted into all possible pairs. For each of the pairs we calculate a conditional probability of one pair element occurrence given the occurrence of other pair element – which is denoted by $P(X_i|X_j)$.

Second order consistency checks allow us to detect another class of anomalies, in which elements already known are used in a previously unknown way. In effect, we have the information about context of feature vector elements.

For probability calculation, we use a Friedman-Singer estimator [8], that provides a way to calculate probability for previously known elements:

$$P(X = i) = \frac{(\alpha + N_i)}{k^0\alpha + N} C \tag{1}$$

as well as probability for previously unknown elements:

$$P(X = i) = \frac{1}{L - k^0}(1 - C) \tag{2}$$

The symbols and their meaning:

α – a virtual value that increases the number of each feature occurrence (it is used as a Dirichlet estimator equivalent)

N_i – number of given feature appearances

N – number of all observations

k^0 – number of different feature values observations

L – number of all possible values of a given feature

C – a scaling variable that represents the relation of probability that an already observed element occurs to probability that a previously unknown element occurs.

Calculation of C coefficient is very computationally expensive. We use a heuristic method [14] which provides following equation:

$$C = \frac{N}{N + L - k^0} \tag{3}$$

This estimation has reasonable computational cost while still providing good results.

A computed probabilities are compared to threshold value, calculated during model creation. If any of the probabilities is smaller than the threshold, then the whole feature vector is denoted as an anomaly and appropriate information is generated.

3.3. Traced system calls selection

Because not all Linux system calls are interesting from the security point of view, a list of traced system calls based on [6] was created. Final list of selected system calls is presented in Table 2 (system call numbers are according to i386 architecture in 2.6.x kernels).

Table 2
Observed system calls

Name	Number	Description
sys_read	3	file read operation
sys_write	4	file write operation
sys_open	5	opens or creates a file
sys_getdents sys_getdents64	141 220	shows directory contents
sys_socketcall	102	socket operations
sys_query_module	167	loaded modules query

Table 2 cont.

sys_setuid sys_getuid	23 24	UID user identifier operations
sys_execve	11	executes binary file
sys_chdir	12	changes directories
sys_fork sys_clone	2 120	creates child processes
sys_ioctl	54	devices controlling
sys_kill	37	sends signals
sys_exit	1	exits from current process
sys_close	6	closes a file descriptor
sys_ptrace	26	allows the parent process to trace and control other processes
sys_setgid sys_getgid	46 47	GID group identifier operations
sys_chmod	15	changes file or directory attributes
sys_lchown sys_chown	16 182	changes owner and group of a given file

Linux source containing list of all system calls and assigned numbers might be found in the `linux/include/asm/unistd.h`.

4. Implementation

4.1. Architecture

Our IDS system consists of two main parts: the *ImSensor* which collects system calls from the kernel and application which collects and analyses information from the sensor.

The main duties of the system are:

- tracing processes activities,
- collecting system calls (and their arguments) information,
- constructing knowledge database which describes "normal" system behaviour,
- evaluation, based on knowledge database, current state of the system.

General diagram of the IDS system has been shown in Figure 1. The system was called **PPIDS**. *ImSensor* collects system calls of traced processes, and puts them into a memory buffer. A block device `/dev/imsensor` is used by the application (which resides in the user space) to read collected system calls and their arguments into database. Those system calls can be later used for anomaly detection.

We have decided to use custom solution (based on [11, 21]) for tracing the system calls rather than `ptrace`. There are actually two main reasons for taking such approach. The first one is that such solution allows us to transform system call arguments according to our needs (for example, if process opens a symbolic link then we convert its name to the file name that it points to).

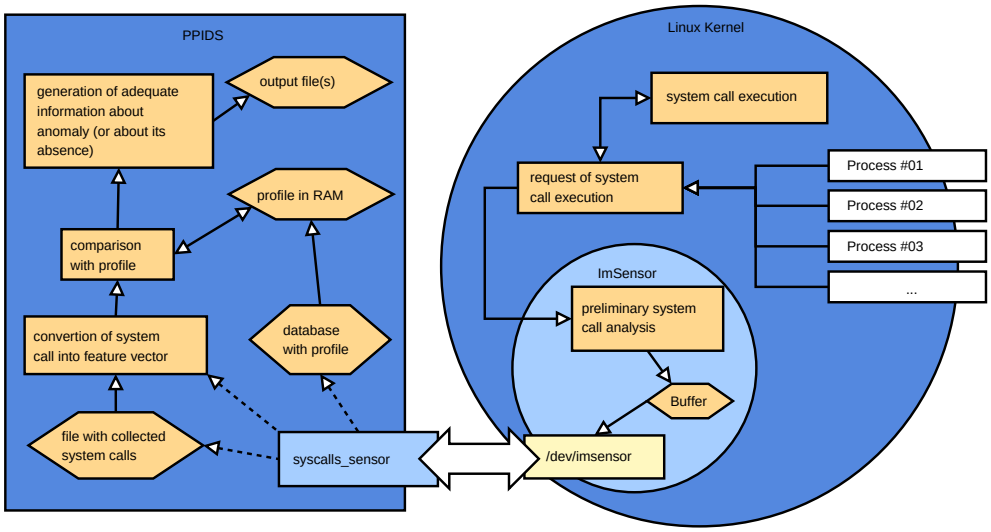

Fig. 1. General overview of our IDS system

The other reason is that it allows us to trace by default all newly created process (not only the child ones) from the very first system call (the "early tracing" feature). The application later might decide if the new process and its system calls are relevant or not.

Our system can perform many different tasks in which we can distinguish two main work modes:

1) **learn** – in which we create basic profile (we assume that this profile contains only correct behaviours),
2) **detect** – in this mode we compare previously created (in the learn mode) profile to the current process behaviour, in order to find anomalies

4.2. Process tracing

While starting PPIDS user defines a list of processes (identified using their names) which are subject of tracing. System calls are analyzed using ID (PID) and name of a process which invokes them. The process is qualified for tracing if either its name is on the list of processes specified for tracing or its PPID (Parent PID) equal to PID of already traced process.

Some processes have a very short life time. This could cause a process to end before PPIDS decide that it should actually be traced. To solve this issue we create a concept of the "early tracing". By default we trace all newly created processes, and keep its system calls information in a buffer. We stop tracing newly created process after 20 seconds or if PPIDS decides that this process should not be traced anymore.

4.3. Collecting the system calls

System calls are collected in the kernel. We have implemented (based on ImSafe[11]/ImSafe2[21]) our own mechanism for collecting system calls (for reasons mentioned above).

The following parts of the kernel were modified:

- the structure `task_struct` which contains information about processes – `linux/kernel/sched.c` file,
- we have added our function into `linux/asm/[i386,...]/entry.S` file, before system call invocation code,
- we have added `/dev/imsensor` device – file `linux/kernel/imsensor.c` – which connects kernel sensor with IDS application.

5. Results

5.1. Test environment

To confirm the effectiveness of our approach to anomaly detection, we have created a special environment inside which all tests took place. It consisted of two PC class computers, connected by Fast Ethernet network.

The PPIDS system, which was the final version of our IDS, was installed on the server PC. All efficiency and performance tests were made on this server. The second computer, which acted as client, was used for connecting to services available on the server. On the client PC we have installed Perl scripts which simulated typical user behaviour.

During the test period both computers worked under Linux OS (Kernel 2.6.12/Fedora Core 4).

5.2. Types of attacks

There are many different system (or application) vulnerabilities, that might be used during an attack. They exists for many reasons, such as programmer's errors or bad software design.

Some of the most popular attack types are:

- **Buffer Overflow Errors** – this attack is using the fact that some widely used programming languages (such as C or C++) do not fully control the memory access. A typical attack scenario is following: programmer creates too small array (or does not control size of the array), so providing the program "special" data (in command line, during execution, as input data, etc.) causes "moving" outside the area meant for the data, into the memory area responsible for program flow. This memory chunk might be accordingly modified and at some point, program starts to execute code provided by the intruder.
- **Denial of Service (DoS)** – attacker tries to slow down or even stop given application or whole system. It might be accomplished by sending specially prepared

(or sometimes – just large enough) data. The attacked object is unable to process such amounts of data, in effect the system resources usage enormously increases.

- **Dictionary attacks** – attacker tries to discover the username and password. The difference between brute-force and dictionary attacks is that dictionary attacks use popular words (and their combinations) rather than random sequences of chars.
- **Viruses and Trojan Horses** – these malicious programs try to get into the machine by presenting themselves as a seemingly harmless application (often sent by an e-mail). When they are run by an inauspicious user, they often perform adverse actions, replicate themselves and send copies to other random machines.
- **Other** – there are many other attack types in existence, which use specific vulnerabilities of a given system. For example – the "SQL Injections" run adverse SQL instructions in weakly secured systems communicating with the SQL database. Or the symbolic-link attacks – which leverage fact that some applications run with administrator rights and they do not check where the link points. In effect – a system file might be overwritten by a normal user.

5.3. Tested applications

To check the effectiveness of our approach we have decided to test our system on some widely used applications, that are prone to different types of attacks. In effect, one of the most important criteria on which applications were chosen were well known security bugs (these vulnerabilities were later used to perform the attacks).

5.3.1. Applications portfolio

The actually chosen applications were following:

- **Prozilla – version 1.3.6-r2** – popular download accelerator available under Linux.This application works by downloading different parts of the chosen file in many threads.
- **CDRDAO – version 1.1.5-10** – application used for burning audio CDs in `DISC AT ONCE` mode. Very often this application has SUID privilege to allow it access to the burning devices. Also, SUID privilege allows running application with `root` user privileges.
- **Apache – version 2.0.47** – the world most popular Web server. This is, as well, the most complicated application we had chosen. During tests Apache served both static HTML pages, and dynamic pages generated by PHP scripts.

To make good evaluation of SysPAD performance it was important to choose diverse set of applications, that would have different security flaws, which could be utilized by attacker in different ways. Some of the applications (Apache) expose network interface, which actually makes possible for anybody to takeover such system. Some (such as CDRDAO) depend on a SUID usage, which poses a danger of getting root privileges by unauthorized user. Finally, some attacks are based on providing malicious

files/mails/web addresses that might be opened by an unwary user (e.g. Prozilla is prone to such security breach).

5.4. Methodology

The first stage of our tests was to collect a set of vectors. In the case of Apache web server we used scripts which simulated typical users behaviour. All other application were run many times, with different arguments. The traced processes were started before the application was run. This allowed us to collect information about application start up, which included initialization of all necessary components like Apache modules. Based on these vectors we created profiles of application-typical behaviour.

The second stage was to collect application behaviour one more time in the same way as we did in the first stage. Vectors, collected in the second stage, were used as behaviour for detection of "false positive anomalies". False positive anomalies are correct behaviours which were classified as anomalies during detection phase.

Next stage was to test our algorithm against different sizes (number of vectors which create profile) of profile – from 1000 to 2000 000 system calls. We have examined creation of the profile and got time of creation as well as diversity of the profile and numbers of false positives anomalies (Fig. 2).

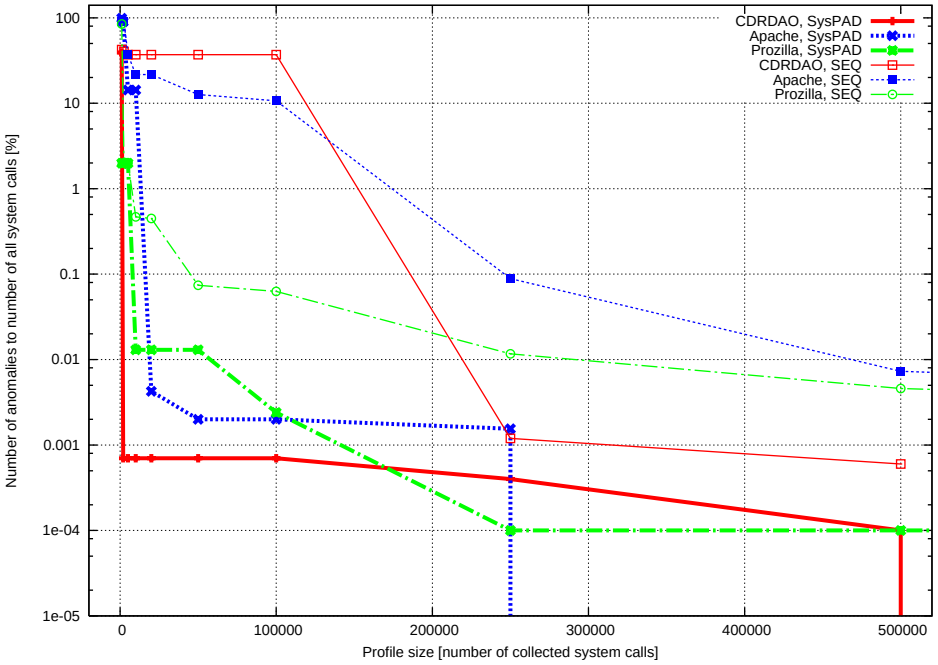

Fig. 2. Number of false anomalies vs. profile size

The last stage of tests was detection of the real anomalies. To achieve this we performed certain numbers of real attacks against test system, using real exploits. We have collected application behaviour during this phase. If attack was successful and the attacker gained access to the system via remote shell, we would perform several typical actions like: `cat /etc/passwd` – obtaining the system users list, `ls /proc` – showing contents of the `/proc` directory, and `w` – checking list of the logged users.

As it might have been expected the Apache web server had the biggest diversity from all tested applications (about 1600 assorted elements). For less complex applications, like Prozilla, CDRDAO or ProFTPd, diversity was much lower (respectively: 250, 400 and 750 assorted elements).

5.5. False positive anomaly detection

False-positive alarms are very dangerous and unwanted events. All methods of anomaly detection, which are based on the process behaviour analysis can generate some false positive alarms. We cannot assume that profile of application is complete, and that it contains all possible application behaviours.

As a base method, to which we compare all our results, we chose SEQ method. For different profile size, which was built from normal application behaviour, we have analysed different set of vectors. All vectors used in this test contain only correct application behaviour.

5.6. Real attacks detection

Figures 3–5 show effectiveness of different kinds of attacks. Each graph shows events probability (on Y axis) and time (X axis). Events which are below threshold are classiffied as anomalies. As it might have been expected, the figures show a vast number of normal events (above the threshold line) and a few "spots" below that are actually detected anomalies.

As we can see, all attacks were eventually detected. Detailed analyzes showed that in Prozilla case (figure 3) we observed the creation of a new process using the `fork` system call and new (previously unseen) application names which were opened by new processes. It was also noticed that new files were opened during this attack. Those new files were not observed during normal application execution. Similar events were spotted during CDRDAO attack. The sophisticated symbolic link attack made that application write into `/etc/cron.d/cdr` file. This was not observed before (during normal execution) and was classified as anomaly.

During DoS attack for Apache web server, our IDS did not detect any different behaviour than in comparison to normal execution, but after system was out of memory Apache invoked new system calls. Those new system calls were calls to swap function and previously unobserved types of sockets communication.

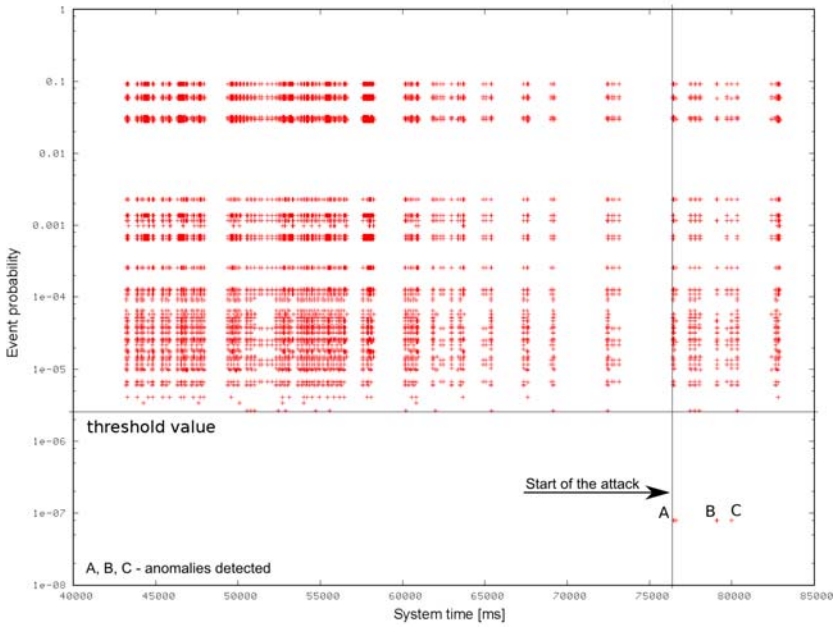

Fig. 3. Prozilla – buffer overflow attack

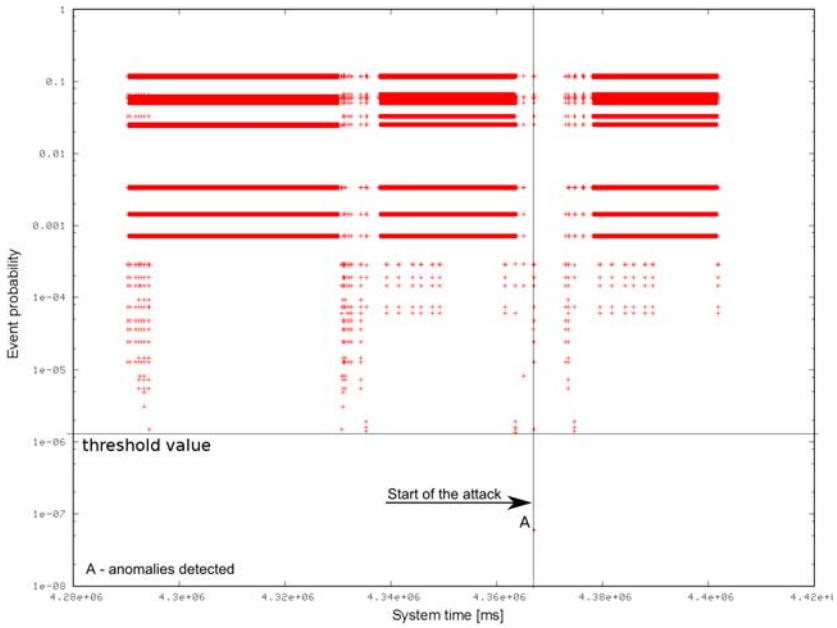

Fig. 4. CDRDAO – symbolic link attack

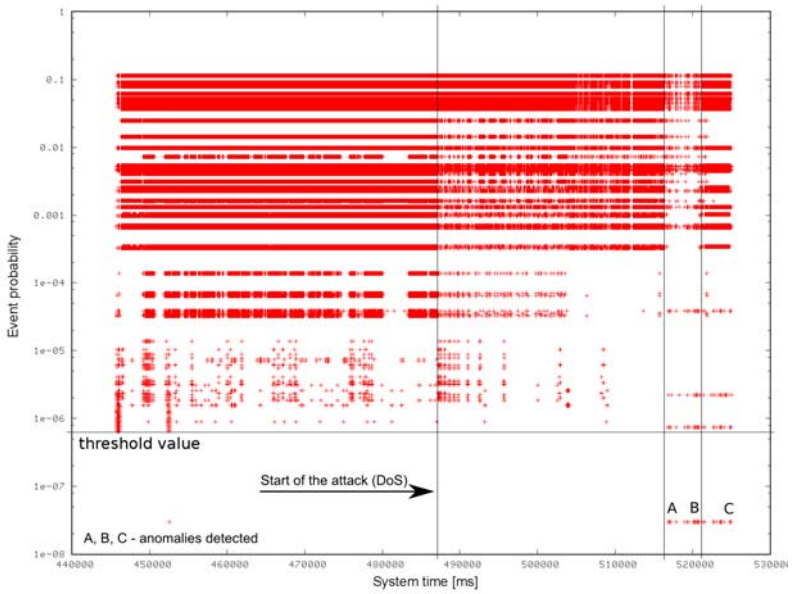

Fig. 5. Apache 2 – DoS attack

6. Summary

We have built our IDS system by extending the PAD algorithm [7]. Our new method – SysPAD uses system calls and their invocation arguments for anomaly detection. The data model uses a set of vectors containing both system calls and their arguments. This data model and PAD algorithm were combined together to create a fast and efficient method allowing effective analysing of application behaviour.

The results show that using SysPAD method is very effective against many different attacks. Its high efficiency comes from analysis of system calls argument. Thanks to this we get a knowledge of not only what a process is doing but also how it is actually done and in which part of the file system. It can be compared to a situation where a policeman knows not only what the criminal is going to do, but also where and how.

Of course it is not possible to build a system without any disadvantages. To allow efficient anomaly detection, our system has to create correct application profiles, which tend to be very complicated in real life programs. We find this as a major disadvantage of any Anomaly-type Intrusion Detecion System. However, we have created some scripts that helped to generate applications behaviour, by calling most of their functions.

At this point, it is expected that the SysPAD will detect most security breaches (assuming it has correct and complete application model). It might have problems

with detecting attacks that cannot be distinguished from normal behaviour on the system calls information basis (such as dictionary attacks). Also, some number of false anomalies might be returned if traced applications are often opening files in random locations (other than user homes or `/tmp` directories).

One area where the SysPAD might fail are the kernel security breaches (as the kernel is not traced). However, it is possible that SysPAD will detect abnormal operations done be previously analyzed application. Thus the attack might be actually detected.

Currently, the SysPAD is not making thorough analysis of socket operations. That is, it doesn't analyze which socket is opened by which service. This is a potential source of useful information.

References

[1] Sekar E., Bendre M., Dhurjati D., Bollineni P.: *A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors*. Proc. of the 2001 IEEE Symposium on Security and Privacy, 2001

[2] Feng H. H., Kolesnikov O. M., Fogla P., Lee W., Gong W.: *Anomaly Detection Using Call Stack Information*. Proc. of the 2003 IEEE Symposium on Security and Privacy, 2003

[3] *Apache webserver 2.0.52 DOS vulnerability – CAN-2004-0942*. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0942`

[4] *Cdrdao Insecure File Handling*. `http://www.securiteam.com/unixfocus/5PP0F1P61I.html`

[5] Warrender Ch., Forrest S., Pearlmutter B.: *Detecting Intrusions Using System Calls: Alternative Data Models*. Proc. of the 1999 IEEE Symposium on Security and Privacy, 1999

[6] Burdach M.: *Detecting Kernel-level Compromises With gdb*. `http://www.securityfocus.com/infocus/1811`

[7] Apap F., Honig A., Hershkop S., Eskin E., Stolfo S.: *Detecting Malicious Software by Monitoring Anomalous Windows Registry Access*. Proc. of Fifth International Symposium of Recent Advances in Intrusion Detection (RAID), 2002

[8] Friedman N., Singer Y.: *Efficient bayesian parameter estimation in large discrete domains*. Neural Information Processing Systems (NIPS 98), 1998

[9] Hershkop S., Bui L. H., Ferster R., Stolfo S. J.: *Host-based Anomaly Detection Using Wrapping File Systems*. Columbia University Tech Report April 2004

[10] Hofmeyera S. A., Forrest S., Somayaji A.: *Intrusion Detection using Sequences of System Calls*. Journal of Computer Security, August 18th, 1998

[11] Eschenauer L. *et al.*: *ImSafe – Host Based Anomaly Detection*. `http://imsafe.sourceforge.net/`

[12] Love R.: *Kernel Locking Techniques*. `http://www.linuxjournal.com/article/5833`

[13] Lee W., Stolfo S. J., Chan P. K.: *Learning patterns from UNIX process execution traces for intrusion detection*. AAAI Workshop on AI Approaches to Fraud Detection and Risk Management, 1997

[14] Heller K. A., Svore K. M., Keromytis A. D., Stolfo S. J.: *One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses*. Proc. of the ICDM Workshop on Data Mining for Computer Security (DMSEC), 2003

[15] Eskin E.: *Probabilistic anomaly detection over discrete records using inconsistency checks*. Columbia University, Computer Science Technical Report, 2002

[16] *Prozilla*, `http://prozilla.genesys.ro/`

[17] Akpolat S.: *Remote Buffer Overflow in Prozilla*. `http://www.securiteam.com/exploits/6W00O2ABPM.html`, October 25th, 2004

[18] SANS Institute: *The Twenty Most Critical Internet Security Vulnerabilities*, `http://www.sans.org/top20`

[19] *Linux Kernel Documentation – SpinLocks*. `http://kernel.org`

[20] *SQLite*, `http://www.sqlite.org`

[21] Dąbrowski P.: *Systemy wykrywające naruszenie bezpieczeństwa w systemie operacyjnym w oparciu o analizę ciągów odwołań systemowych*. Kraków, Katedra Informatyki AGH, September 2004

[22] Mitnick K.: *The Art of Deception: Controlling the Human Element of Security*. 1st edition, John Wiley & Sons, 2001, ISBN 978-0471237129

[23] Bovet D. P., Cesati M.: *Understanding the Linux Kernel*. 2nd Edition, O'Reilly Media, Inc., 2002, ISBN 978-0596002138